The VLISP Flattener

J. D. Guttman L. G. Monk J. D. Ramsdell
W. M. Farmer V. Swarup

The MITRE Corporation*

M92B094
September 1992

*Authors' address: The MITRE Corporation, 202 Burlington Road, Bedford MA, 01730-1420.

Abstract

The Verified Programming Language Implementation project has developed a formally verified implementation of the Scheme programming language. This report documents the flattener, which linearizes a tree-structured byte-code. It contains detailed proofs that the operational semantics of the flattened output matches the operational semantics of the input.

Contents

1 Introduction

The primary purpose of this paper is to present and justify the VLISP passage from tree-structured code to code which is essentially linear, using an operational semantics. As the flattener is a convenient mid-point in the overall VLISP program [2], this paper also serves as a reference, giving some official definitions for other parts of the program. Specifically, this paper presents:

(1) the VLISP approach to operational semantics via state machines,

(2) the syntax and operational semantics for the VLISP Tabular Byte Code (TBC),

(3) the syntax and operational semantics for the VLISP Flattened Byte Code (FBC),

(4) the *flattener* algorithm for passing from TBC programs to FBC programs, and

(5) an extension of the flattener to a map from initial TBC states to initial FBC states that produce the same answers, with a proof of this fact.

We define (tabular) byte code state machines and flattened byte code state machines below as kinds of deterministic state transition machines with concrete states, in accordance with a general formalism for state machines and their computations. This formalism allows for programs to result in answers, as in the denotational semantics, and also for a later, more refined view of programs as operators that take input streams to output streams. The byte code machines' states are vectors of syntactic objects of expediently augmented languages that are formed from the tabular byte code and the flattened byte code by adding syntactic constructors for terms for environments, closures, continuations, etc. Each of the various kinds of state transitions is motivated by a possible combination of equational reductions in the denotational semantics of the underlying byte code.

1.1 Notation

We identify natural numbers with finite von Neumann ordinals, so that each natural number is actually the set of all smaller natural numbers. A finite sequence is a function with a natural number as its domain, which is the same as its length. Thus, in our usage, when s is a finite sequence, $s(2)$ is the third element of s, if s has length at least three, and is undefined if the length of s is strictly less than three. We use angle brackets to indicate sequence formation, with spaces (and sometimes, for clarity, commas) separating elements, and reserve ordinary parentheses for various other traditional uses.

Asterisk is usually used as an operator for finite sequence formation, thus X^* means the set of all finite sequences from X, but asterisks are also sometimes just used as mnemonics in variable names for variables that range over sets of finite sequences.

In contrast to some uses of BNF, there is no implicit concatenation of finite sequence terms inside angle brackets; for instance, every element of $\langle A\ B^*\ C^*\rangle$, which means the same as $\langle A, B^*, C^*\rangle$, is a sequence of length exactly three.

We will also use the following notation:

$\#s$	the length of sequence s
$s \frown t$	the concatenation of sequences s and t
$x :: s$	$\langle x\rangle \frown s$ ("mathematical CONS")
$s \dagger k$	the result of dropping the first k members from sequence s
$s \ddagger k$	the sequence of only the first k members of s
$a \to b, c$	if a then b else c

2 State Transition Machines in General

We choose one convenient way of pinning down details among many which would suffice. First we define (possibly non-deterministic) *state transition machines*, these are septuples

$$\langle \textit{states},\ \textit{halt states},\ \textit{inputs},\ \textit{null-input},\ \textit{outputs},\ \textit{null-output},\ \textit{actions} \rangle,$$

such that

(a) *states*, *inputs*, and *outputs* are disjoint sets with *halt states* $\subseteq$ *states*, *null-input* $\in$ *inputs*, and *null-output* $\in$ *outputs*,

(b) *actions* $\subseteq$ (*states* $\times$ *inputs*) $\times$ (*states* $\times$ *outputs*), and

(c) For no pair $\langle s,\ i \rangle$ in the domain of actions is $s \in$ *halt states.*

Given such a machine M, let C be a non-empty, finite or infinite sequence of elements of *states* $\times$ *inputs* $\times$ *outputs.* Then C_S (the *state history*), C_I (the *input history*), and C_O (the *output history*), of C are, respectively, the derived sequences of *states*, *inputs*, and *outputs* components of C. C is a *computation* if $C_O(0) =$ *null-output*, and, for every i such that $C(i+1)$ is defined,

$$\langle \langle C_S(i), C_I(i) \rangle,\ \langle C_S(i+1), C_O(i+1) \rangle \rangle \in \textit{actions}.$$

We think of a transition according to an action as taking a pair of a current state and a current input and producing a next state and a next output; the initial output should be null.

Call a state-input pair *proceedable* if it is in the domain of *actions.* A computation is *maximal* if it is infinite or its last state and last input comprise a non-proceedable pair. A finite, maximal computation is *successful* if its last state is in *halt states*; other finite, maximal computations are *erroneous.* We also call a state-input pair *erroneous* if it is not proceedable, but the state is not in *halt states*; thus a computation C is erroneous if and only if, for some i, $\langle C_S(i), C_I(i) \rangle$ is erroneous. We allow for the possibility that some states are erroneous when paired with some inputs and not with others.

A computation is *inputless* if every element of its input history is null; it is *outputless* if every element of its output history is null; it is *pure* if it is both inputless and outputless. The *output* of a computation is the sequence of non-null elements of its output history. The input history and output history of a computation are not to be thought of as identical to the input stream and output stream. The connection depends on the run-time behavior of the program and requires definition by recursion on computations, using the additional notions of read and write operations. If *actions* is actually a function, then the machine is *deterministic*.

3 Tabular Byte Code State Machines

This section presents first the syntax of the Tabular Byte Code and an expansion of the Tabular Byte Code called the Augmented Byte Code. The Augmented Byte Code is then used to define the states of Tabular Byte Code State Machines. Finally, the action relation of these machines is presented as a union of subfunctions called rules. We may also call them ABC rules, or even TBC rules, to distinguish from the rules for the flattened code, to be defined later.

3.1 Tabular Byte Code

The Tabular Byte Code (or TBC) provides crude tables (just sequences of entries) that allow the code in templates to refer to constants, global variables (identifiers), and other templates indirectly by indexing. Otherwise it is almost exactly the same as the BBC described in the accompanying Vlisp report on the Byte Code Compiler [1].

The tokens of the TBC are the same as those of the BBC, with the addition of `constant`, and `global-variable`, and with `lap` being replaced by `template`. We adapt and extend the variable conventions of BBC. Thus z stands here for a (TBC) neutral instruction (or the class of neutral instructions), and so on for all of the TBC syntactic classes, including two new classes: d is used for (TBC) *table entries* (or *template literals*), and e for (TBC) *template tables*. The defining productions (given in Table 1) are very similar to the ones given for BBC, but note that:

- in a template, the block now precedes an important table, instead of following an unimportant constant, and
- `literal`, `closure`, `global`, and `set-global!` take integer arguments here.

We will use n-like variables for natural numbers, i-like variables for identifiers, and c-like variables for constants. Similarly for the classes defined by the grammar, with

z for (TBC) *neutral instructions*,

m for (TBC) (*machine*) *instructions*,

b for (TBC) *closed instruction lists* (conjecturally $<$ Eng. *block*),

y for (TBC) *open instruction lists*,

$$
\begin{array}{lll}
z & ::= & \langle \texttt{unless-false}\ y_1\ y_2 \rangle \\
 & | & \langle \texttt{literal}\ n \rangle \ \mid\ \langle \texttt{closure}\ n \rangle \\
 & | & \langle \texttt{global}\ n \rangle \ \mid\ \langle \texttt{local}\ n_1\ n_2 \rangle \\
 & | & \langle \texttt{set-global!}\ n \rangle \ \mid\ \langle \texttt{set-local!}\ n_1\ n_2 \rangle \\
 & | & \langle \texttt{push} \rangle \ \mid\ \langle \texttt{make-env}\ n \rangle \\
 & | & \langle \texttt{make-rest-list}\ n \rangle \ \mid\ \langle \texttt{unspecified} \rangle \\
 & | & \langle \texttt{checkargs=}\ n \rangle \ \mid\ \langle \texttt{checkargs>=}\ n \rangle \\
 & | & \langle i \rangle \\
m & ::= & z \ \mid\ \langle \texttt{return} \rangle \ \mid\ \langle \texttt{call}\ n \rangle \ \mid\ \langle \texttt{unless-false}\ b_1\ b_2 \rangle \\
 & | & \langle \texttt{make-cont}\ w_1\ n \rangle \\
b & ::= & \langle\langle \texttt{return} \rangle\rangle \ \mid\ \langle\langle \texttt{call}\ n \rangle\rangle \ \mid\ \langle\langle \texttt{unless-false}\ b_1\ b_2 \rangle\rangle \\
 & | & \langle \texttt{make-cont}\ b_1\ n \rangle :: b_2 \ \mid\ z :: b_1 \\
y & ::= & \langle \texttt{make-cont}\ y_1\ n \rangle :: b \ \mid\ \langle \texttt{make-cont}\ \langle\rangle\ n \rangle :: b \ \mid\ z :: y_1 \ \mid\ \langle z \rangle \\
w & ::= & b \ \mid\ y \\
d & ::= & \langle \texttt{constant}\ c \rangle \ \mid\ \langle \texttt{global-variable}\ i \rangle \ \mid\ t \\
e & ::= & d^* \\
t & ::= & \langle \texttt{template}\ b\ e \rangle
\end{array}
$$

Table 1: Grammar for the Tabular Byte Code

> w for (TBC) *(general) instruction lists*,
>
> d for (TBC) *template literals*,
>
> e for (TBC) *template literal tables*, and
>
> t for (TBC) *templates*.

In the flattener algorithm, we treat open and closed instruction lists differently. In particular, in an `unless-false` with open instruction lists y_1 and y_2, the flattener must insert an unconditional jump at the end of the true branch, while this would be redundant in the case of an `unless-false` with closed instruction lists. Instructions of the form $\langle i \rangle$ are used only for "primop" identifiers (like `%%cons`) associated with a small set of denotationally specified Scheme primitives.

Given an open instruction list y and any instruction list w, we will sometimes need to refer to their *open-adjoin*, written $y \smile w$ and defined as follows:

> $\langle z \rangle \smile w = z :: w;$
>
> $(z :: y) \smile w = z :: (y \smile w);$

$(\langle \texttt{make-cont}\ y\ n\rangle :: b)^\smile w = \langle \texttt{make-cont}\ y^\smile w\ n\rangle :: b$; and

$(\langle \texttt{make-cont}\ \langle\rangle\ n\rangle :: b)^\smile w = \langle \texttt{make-cont}\ w\ n\rangle :: b$.

It may help to note, first, that a closed instruction list can contain no instruction of the form $\langle \texttt{make-cont}\ y\ n\rangle$ or $\langle \texttt{make-cont}\ \langle\rangle\ n\rangle$, and, hence, that an open instruction list is either a list of neutral instructions or contains exactly one instruction of this form. Open-adjoin proceeds recursively inwards from its first argument through the code lists of such instructions until it reaches one which is just a (possibly empty) list of neutral instructions and then it concatenates w to the "open" end of that list.

Lemma 1 Properties of open-adjoin. *For all y, w, y_1, and b,*

(1) $y^\smile w$ is well-defined by the above recursion and is an instruction list;

(2) $y^\smile y_1$ is open, and $y^\smile b$ is closed; and

(3) $y^\smile(y_1{}^\smile w) = (y^\smile y_1)^\smile w$ (associativity).

Proof. Observe first that the four expressions used to define productions of open instruction lists define injective functions (of the variables occurring in them) with disjoint ranges, so the definition of $y^\smile w$ is a legitimate recursion on y. The rest of (1) and (2) follow by easy inductions on y. With these established, (3) follows by induction on y, with an inner induction on y_1. QED.

3.2 The Augmented Byte Code Language

The Augmented Byte Code (ABC) provides the syntactic objects that are the components of the state of the byte code state machine. It is formed by expanding the Tabular Byte Code defined above to allow terms for environments, closures, continuations, etc. As we will eventually use the byte code state machine, these constructors do not occur in the code coming from Scheme programs that it runs, but are put in its initial store or are generated in the course of its computation.

The new ABC tokens are locations (which are actually the same as natural numbers, but which for clarity we indicate by l-like variables when used as locations) and the following *constructors*:
CLOSURE, ESCAPE, MUTABLE-PAIR, VECTOR, STRING,
UNDEFINED, NOT-SPECIFIED, EMPTY-ENV, ENV, HALT, and CONT.

The new ABC syntactic categories to be defined by BNF are

v for (ABC) *values*,
a for (ABC) *argument stacks*
u for (ABC) *environments*,
k for (ABC) *continuations*, and
s for (ABC) *stores*.

All of the categories (templates, blocks etc.) and productions given for TBC are included here, but the extensions of the syntax we are making here do not feed back into the recursive construction of the old categories, so their extensions in the ABC are exactly the same as in the TBC. The new productions are

$$
\begin{array}{lcl}
v & ::= & c \mid \langle \text{CLOSURE}\ t\ u\ l \rangle \mid \langle \text{ESCAPE}\ k\ l \rangle \\
 & \mid & \langle \text{MUTABLE-PAIR}\ l_1\ l_2 \rangle \mid \langle \text{STRING}\ l^* \rangle \mid \langle \text{VECTOR}\ l^* \rangle \\
 & \mid & \text{NOT-SPECIFIED} \mid \text{UNDEFINED} \\
a & ::= & v^* \\
u & ::= & \text{EMPTY-ENV} \mid \langle \text{ENV}\ u\ l^* \rangle \\
k & ::= & \text{HALT} \mid \langle \text{CONT}\ t\ b\ a\ u\ k \rangle \\
s & ::= & v^*
\end{array}
$$

3.3 States and Actions

The states of a byte code state machine are the sequences of the form

$$\langle t,\ b,\ v,\ a,\ u,\ k,\ s \rangle,$$

with the abuse of notation that b is allowed to be $\langle\rangle$. The components of a state are called, in order, its *template*, *code*, *value*, *argument stack*, *environment*, *continuation*, and *store*, and we may informally speak of them as being held in registers. The *halt states* are the states such that $b = \langle\rangle$.

The rules depend on a parameter, called *globals*, which can be thought of as a kind of environment, ultimately to be built by the compiler; officially it is a function from some finite set of identifiers into locations. An alternative would be to make *globals* a slightly less concrete component of the machine state that is not changed by any action.

We present *actions* as the union of subfunctions called (*action*) *rules*. The action rules are functions from disjoint subsets of *states* $\times$ *inputs* into *states* $\times$ *outputs*. A rule is *pure* if all pairs in its domain have input equal to *null-input*, and all pairs in its range have output equal to *null-output*; other rules are called *port* rules. We will only specify pure rules at this level of VLISP, and some correctness assertions assume that all state transitions are according to one of the rules explicitly specified here.

3.3.1 Auxiliary Functions

As # yields the length of a list, given any ABC store s, $\#s$ is the domain of s and is also the least l such that $s(l)$ is not defined. $s\frown x$ is another store if x is a sequence of values. Also, $s + \{l \mapsto v\}$ means the function whose domain is $\#s \cup \{l\}$ and which takes on the value v at l; this is a store if and only if $l \leq \#s$; it may or may not be compatible with s.

The ternary function *env-reference* is defined recursively for some triples of an ABC environment and two natural numbers by

1. $\textit{env-reference}\,(\langle \text{ENV}\ u\ l^* \rangle,\ 0,\ n_1 + 1) = l^*(n_1)$, and
2. $\textit{env-reference}\,(\langle \text{ENV}\ u\ l^* \rangle,\ n_0 + 1,\ n_1) = \textit{env-reference}\,(u,\ n_0,\ n_1)$

One might say that *env-reference* is 1-based in its last argument. Note that $\textit{env-reference}\,(\text{EMPTY-ENV},\ n_0,\ n_1)$ is never defined.

To build increasingly complex ABC environment terms, the ternary function *add-layer* is useful. For $n_1 > 0$,

$$\textit{add-layer}(u,\ n_0,\ n_1) = \langle \text{ENV}\ u\ \langle n_0\ \ldots\ (n_0{+}n_1{-}1)\rangle\rangle,$$

and $\textit{add-layer}(u,\ n_0,\ 0) = \langle \text{ENV}\ u\ \langle\rangle\rangle$.

For `make-rest-list` we need two functions: *mrl-value* and *mrl-store*. Both take four arguments: a natural number, a value, an argument stack, and a store; both assume that their first argument is at most the length of their third argument; and both are defined by similar recursions:

$$\begin{aligned}
&\textit{mrl-value}\,(0,\ v,\ a,\ s) = v, \text{ and}\\
&\textit{mrl-value}\,(n+1,\ v,\ a,\ s) =\\
&\quad \textit{mrl-value}\,(n, \langle \text{MUTABLE-PAIR}\ \#s\ (\#s+1)\rangle,\ a \dagger 1,\ s\frown\langle a(0), v\rangle).
\end{aligned}$$

$$\begin{aligned}
&\textit{mrl-store}\,(0,\ v,\ a,\ s) = s, \text{ and}\\
&\textit{mrl-store}\,(n+1,\ v,\ a,\ s) =\\
&\quad \textit{mrl-store}\,(n, \langle \text{MUTABLE-PAIR}\ \#s\ (\#s+1)\rangle,\ a \dagger 1,\ s\frown\langle a(0), v\rangle).
\end{aligned}$$

Note that $\mathit{mrl\text{-}value}\,(n,\ v,\ a,\ s)$ does not really depend on anything about a and s other than their lengths, that is, there is a function $\mathit{mrl\text{-}value}'$ such that, for all $n,\ v,\ a$, and s,

$$\mathit{mrl\text{-}value}\,(n,\ v,\ a,\ s) = \mathit{mrl\text{-}value}'\,(n,\ v,\ \#a,\ \#s).$$

We will also need a function *app-stack* for constructing a list of arguments for `%%apply`. The arguments to *app-stack* are a value, an argument stack, and a store. It uses the stack in the same "reverse" order as the compiler does when it pushes arguments when constructing calls. This is the first time we have needed a function that actually must look inside Scheme constants – it needs to know about `immutable-pair`, as well as about the Scheme constant *null*. The recursive definition has three cases:

$$\begin{aligned}
&\mathit{app\text{-}stack}\,(\mathit{null},\ a,\ s) = a,\\
&\mathit{app\text{-}stack}\,(\langle \text{MUTABLE-PAIR}\ l_1\ l_2\rangle,\ a,\ s) =\\
&\qquad \mathit{app\text{-}stack}\,(s(l_2),\ s(l_1) :: a,\ s), \text{ and}\\
&\mathit{app\text{-}stack}\,(\langle \texttt{immutable-pair}\ v_1\ v_2\rangle,\ a,\ s) =\\
&\qquad \mathit{app\text{-}stack}\,(v_2,\ v_1 :: a,\ s).
\end{aligned}$$

The Primitive ADD rule allows an argument stack to have arbitrarily many arguments, including none. For it we use the auxiliary function *n-ary-sum*, which is defined at an argument a if and only if a is a list of numbers, in which case its value is just the sum of all elements of a.

3.3.2 Presentation Format for Pure Rules

For each pure rule we give a name, one or more conditions determining when the rule is applicable (and possibly introducing new locally bound variables for later use), and a specification of the new values of some registers. Often the domain is specified by equations giving "the form" of certain registers, especially the code. In all specifications the original values of the various registers are designated by conventional variables used exactly as in the above definition of a state: $t,\ b,\ v,\ a,\ u,\ k$, and s. Call these the original register variables. The new values of the registers are indicated by the same variables with primes attached: $t',\ b',\ v',\ a',\ u',\ k'$, and s'. Call these the new register variables. New register variables occur only as the left hand sides of equations specifiying new register values. Registers for which no new value is given are tacitly asserted to remain unchanged. Additionally, we use e for $t(2)$; thus e must always be a template table no matter what the original state is, just because it is a valid state. Input and output must both be null.

It may help to be more precise about the use of local bindings derived from pattern matching. The domain conditions may involve the original register variables and may introduce new variables (other than e and not among the new or old register variables). If we call these new, "auxiliary" variables $x_1, \ldots, x_j$, then the domain conditions define a relation of $j + 7$ places

$$(\dagger) \quad R(t, b, v, a, u, k, s, x_1, \ldots, x_j).$$

The domain condition really is this: the rule can be applied in a given state if there exist $x_1, \ldots, x_j$ such that ($\dagger$). Furthermore, in the change specifications we assume for these auxiliary variables a local binding such that ($\dagger$). Independence of the new values on the exact choice (if there is any choice) of the local bindings will be unproblematic.

3.3.3 Return-like Rules

Rule 1: **Return-Halt**
Domain conditions:
$b = \langle\langle \texttt{return} \rangle\rangle$
$k = \text{HALT}$
Changes:
$b' = \langle\rangle$

Rule 2: **Return**
Domain conditions:
$b = \langle\langle \texttt{return} \rangle\rangle$
$k = \langle \text{CONT } t_1 \ b_1 \ a_1 \ u_1 \ k_1 \rangle$
Changes:
$t' = t_1$
$b' = b_1$
$a' = a_1$
$u' = u_1$
$k' = k_1$

Rule 3: **Call**
Domain conditions:
$b = \langle\langle \texttt{call}\ \#a\rangle\rangle$
$v = \langle \text{CLOSURE}\ t_1\ u_1\ l_1\rangle$
$t_1 = \langle \texttt{template}\ b_1\ e_1\rangle$
Changes:
$t' = t_1$
$b' = b_1$
$u' = u_1$

Rule 4: **Escape-Halt**
Domain conditions:
$b = \langle\langle \texttt{call}\ 1\rangle\rangle$
$v = \langle \text{ESCAPE HALT}\ l\rangle$
Changes:
$b' = \langle\rangle$

Rule 5: **Escape**
Domain conditions:
$b = \langle\langle \texttt{call}\ 1\rangle\rangle$
$v = \langle \text{ESCAPE}\ \langle \text{CONT}\ t_1\ b_1\ a_1\ u_1\ k_1\rangle\ l\rangle$
$a = \langle v_1\rangle$
Changes:
$t' = t_1$
$b' = b_1$
$v' = v_1$
$a' = a_1$
$u' = u_1$
$k' = k_1$

3.3.4 Branch Rules

Rule 6: **Closed Branch/True**
Domain conditions:
$b = \langle\langle \texttt{unless-false}\ b_1\ b_2 \rangle\rangle$
$v \neq false$
Changes:
$b' = b_1$

Rule 7: **Closed Branch/False**
Domain conditions:
$b = \langle\langle \texttt{unless-false}\ b_1\ b_2 \rangle\rangle$
$v = false$
Changes:
$b' = b_2$

Rule 8: **Open Branch/True**
Domain conditions:
$b = \langle \texttt{unless-false}\ y_1\ y_2 \rangle :: b_1$
$v \neq false$
Changes:
$b' = y_1 {}^\smile b_1$

Rule 9: **Open Branch/False**
Domain conditions:
$b = \langle \texttt{unless-false}\ y_1\ y_2 \rangle :: b_1$
$v = false$
Changes:
$b' = y_2 {}^\smile b_1$

3.3.5 Other Basic Rules

Rule 10: **Make Continuation**
Domain conditions:
  $b = \langle \texttt{make-cont}\ b_1\ \#a \rangle :: b_2$
Changes:
  $b' = b_2$
  $a' = \langle\rangle$
  $k' = \langle \textsc{cont}\ t\ b_1\ a\ u\ k \rangle$

Rule 11: **Literal**
Domain conditions:
  $b = \langle \texttt{literal}\ n \rangle :: b_1$
  $e(n) = \langle \texttt{constant}\ c \rangle$
Changes:
  $b' = b_1$
  $v' = c$

Rule 12: **Closure**
Domain conditions:
  $b = \langle \texttt{closure}\ n \rangle :: b_1$
  $e(n) = \langle \texttt{template}\ b_2\ e_2 \rangle$
Changes:
  $b' = b_1$
  $v' = \langle \textsc{closure}\ e(n)\ u\ \#s \rangle$
  $s' = s \frown \langle \textsc{not-specified} \rangle$

Rule 13: **Global**
Domain conditions:
  $b = \langle \texttt{global}\ n \rangle :: b_1$
  $e(n) = \langle \texttt{global-variable}\ i \rangle$
  $globals(i) = l$
  $v_1 = s(l)$
  $v_1 \neq \textsc{undefined}$
Changes:
  $b' = b_1$
  $v' = v_1$

Rule 14: **Set Global**
Domain conditions:
$b = \langle \texttt{set-global!}\ n \rangle :: b_1$
$e(n) = \langle \texttt{global-variable}\ i \rangle$
$l = \mathit{globals}(i)$
$l \leq \#s$
Changes:
$b' = b_1$
$v' = \text{NOT-SPECIFIED}$
$s' = s + \{l \mapsto v\}$

Rule 15: **Local**
Domain conditions:
$b = \langle \texttt{local}\ n_1\ n_2 \rangle :: b_1$
$l = \mathit{env\text{-}reference}(u,\ n_1,\ n_2)$
$v_1 = s(l)$
$v_1 \neq \text{UNDEFINED}$
Changes:
$b' = b_1$
$v' = v_1$

Rule 16: **Set Local**
Domain conditions:
$b = \langle \texttt{set-local!}\ n_1\ n_2 \rangle :: b_1$
$l = \mathit{env\text{-}reference}(u,\ n_1,\ n_2)$
$l \leq \#s$
Changes:
$b' = b_1$
$v' = \text{NOT-SPECIFIED}$
$s' = s + \{l \mapsto v\}$

Rule 17: **Push**
Domain Conditions:
$b = \langle \texttt{push} \rangle :: b_1$
Changes:
$b' = b_1$
$a' = v :: a$

Rule 18: **Make Environment**
Domain Conditions:
$b = \langle \texttt{make-env}\ \#a \rangle :: b_1$
Changes:
$b' = b_1$
$a' = \langle\rangle$
$u' = \mathit{add\text{-}layer}(u,\ \#s,\ \#a)$
$s' = s \frown a$

Rule 19: **Make Rest List**
Domain Conditions:
$b = \langle \texttt{make-rest-list}\ n_1 \rangle :: b_1$
$n_1 + n_2 = \#a$
Changes:
$b' = b_1$
$v' = \mathit{mrl\text{-}value}\,(n_2,\ \mathit{null},\ a,\ s)$
$a' = a \dagger n_2$
$s' = \mathit{mrl\text{-}store}\,(n_2,\ \mathit{null},\ a,\ s)$

Rule 20: **Unspecified**
Domain Conditions:
$b = \langle \texttt{unspecified} \rangle :: b_1$
Changes:
$b' = b_1$
$v' = $ NOT-SPECIFIED

Rule 21: **Check Args =**
Domain Conditions:
$b = \langle \texttt{checkargs=}\ n \rangle :: b_1$
$\#a = n$
Changes:
$b' = b_1$

Rule 22: **Check Args >=**
Domain Conditions:
$b = \langle \texttt{checkargs>=}\ n \rangle :: b_1$
$\#a \geq n$
Changes:
$b' = b_1$

3.3.6 Primitive Operation Rules

Rule 23: **Primitive CWCC**
Domain Conditions:
 $b = \langle \texttt{\%\%cwcc} \rangle :: b_1$
 $a = \langle v_1 \rangle$
 $v_1 = \langle \textsc{closure}\ t_1\ u_1\ l_1 \rangle$
Changes:
 $t' = t_1$
 $b' = t_1(1)$
 $v' = v_1$
 $a' = \langle \langle \textsc{escape}\ k\ \#s \rangle \rangle$
 $u' = u_1$
 $s' = s \frown \langle \textsc{not-specified} \rangle$

Rule 24: **Primitive CWCC-Escape**
Domain Conditions:
 $b = \langle \texttt{\%\%cwcc} \rangle :: b_0$
 $a = \langle \langle \textsc{escape}\ \langle \textsc{cont}\ t_1\ b_1\ a_1\ u_1\ k_1 \rangle\ l_1 \rangle \rangle$
Changes:
 $t' = t_1$
 $b' = b_1$
 $v' = \langle \textsc{escape}\ k\ \#s \rangle$
 $a' = a_1$
 $u' = u_1$
 $k' = k_1$
 $s' = s \frown \langle \textsc{not-specified} \rangle$

Rule 25: **Primitive CWCC-Escape-Halt**
Domain Conditions:
 $b = \langle \texttt{\%\%cwcc} \rangle :: b_1$
 $a = \langle \langle \textsc{escape halt}\ l \rangle \rangle$
Changes:
 $b' = \langle \rangle$
 $v' = \langle \textsc{escape}\ k\ \#s \rangle$

Rule 26: **Primitive Cons**
Domain Conditions:
$b = \langle \texttt{\%\%cons} \rangle :: b_1$
$a = \langle v_1\ v_2 \rangle ^\frown a_1$
Changes:
$b' = b_1$
$v' = \langle \textsc{mutable-pair}\ \#s\ (1 + \#s) \rangle$
$a' = \langle \rangle$
$s' = s ^\frown \langle v_2\ v_1 \rangle$

Rule 27: **Primitive Car-Immutable Pair**
Domain Conditions:
$b = \langle \texttt{\%\%car} \rangle :: b_1$
$a = \langle \texttt{immutable-pair}\ c_1\ c_2 \rangle :: a_1$
Changes:
$b' = b_1$
$v' = c_1$
$a' = \langle \rangle$

Rule 28: **Primitive Car-Mutable Pair**
Domain Conditions:
$b = \langle \texttt{\%\%car} \rangle :: b_1$
$a = \langle \textsc{mutable-pair}\ l_1\ l_2 \rangle :: a_1$
$v_1 = s(l_1)$
Changes:
$b' = b_1$
$v' = v_1$
$a' = \langle \rangle$

Rule 29: **Primitive Set-Car!**
Domain Conditions:
$b = \langle \texttt{\%\%set-car!} \rangle :: b_1$
$a = \langle v_1\ \langle \textsc{mutable-pair}\ l_1\ l_2 \rangle \rangle ^\frown a_1$
$l_1 < \#s$
Changes:
$b' = b_1$
$v' = \textsc{not-specified}$
$a' = \langle \rangle$
$s' = s + \{l_1 \mapsto v_1\}$

Rule 30: **Primitive Apply-Closure**
Domain Conditions:

$b = \langle \texttt{\%\%apply} \rangle :: b_1$
$a = \langle v_1 \rangle ^\frown a_1 {}^\frown \langle v_2 \rangle$
$v_2 = \langle \textsc{closure}\ t_1\ u_1\ l_1 \rangle$
$t_1 = \langle \texttt{template}\ b_2\ e_2 \rangle$
$a_2 = \mathit{app\text{-}stack}(v_1,\ a_1,\ s)$

Changes:

$t' = t_1$
$b' = b_2$
$v' = v_2$
$a' = a_2$
$u' = u_1$

Rule 31: **Primitive Apply-Escape**
Domain Conditions:

$b = \langle \texttt{\%\%apply} \rangle :: b_1$
$a = \langle v_1 \rangle ^\frown a_1 {}^\frown \langle v_2 \rangle$
$v_2 = \langle \textsc{escape}\ k_1\ l_1 \rangle$
$k_1 = \langle \textsc{cont}\ t_2\ b_2\ a_2\ u_2\ k_2 \rangle$
$\langle v_3 \rangle = \mathit{app\text{-}stack}(v_1,\ a_1,\ s)$

Changes:

$t' = t_2$
$b' = b_2$
$v' = v_3$
$a' = a_2$
$u' = u_2$
$k' = k_2$

Rule 32: **Primitive Apply-Escape-Halt**
Domain Conditions:

$b = \langle \texttt{\%\%apply} \rangle :: b_1$
$a = \langle v_1 \rangle ^\frown a_1 {}^\frown \langle v_2 \rangle$
$v_2 = \langle \textsc{escape halt}\ l_1 \rangle$

Changes:

$b' = \langle \rangle$
$v' = v_2$

Rule 33: **Primitive Eqv**
Domain Conditions:
$b = \langle \texttt{\%\%eqv} \rangle :: b_1$
$a = \langle v_1 \ v_2 \rangle ^\frown a_1$
Changes:
$b' = b_1$
$v' = ((v_1 = v_2) \rightarrow true, false)$
$a' = \langle \rangle$

Rule 34: **Primitive Add**
Domain Conditions:
$b = \langle \texttt{\%\%add} \rangle :: b_1$
$m = n\text{-}ary\text{-}sum\,(a)$
Changes:
$b' = b_1$
$v' = m$
$a' = \langle \rangle$

4 Flattened Byte Code State Machines

The purpose of this section is to present the FBC and give it an (operational) semantics in a style very similar to the presentation above of the operational semantics of the TBC.

4.1 Syntax of the Flattened Byte Code Language

The Flattened Byte Code (FBC) is a modification of the Tabular Byte Code that uses exclusively unnested linear sequences of tokens for the code part of its templates. Instead of the TBC conditional instruction `unless-false`, FBC has `jumpf` and `jump`, both of which take a pair of numeric operands that together indicate an offset to another location in the instruction sequence. The syntax of `make-cont` has been altered so that it too can use numeric offsets.

Conventional base variables for the FBC syntactic classes to be defined by BNF are:

> t for *(*FBC*) templates*,
> e for *(*FBC*) tables*,
> o for *(*FBC*) table entries* (or *template literals*),
> m for *(*FBC*) instructions* (an operator with its operands), and
> w for *(*FBC*) code sequences* (concatenations of instructions).

The BNF definitions of these classes follows.

$$
\begin{array}{rcl}
t & ::= & \langle \texttt{template}\; w\; e \rangle \\
e & ::= & o^* \\
o & ::= & \langle \texttt{constant}\; c \rangle \mid \langle \texttt{global-variable}\; i \rangle \mid t \\
w & ::= & \langle\rangle \mid m ^\frown w \\
m & ::= & \langle \texttt{call}\; n \rangle \mid \langle \texttt{return} \rangle \mid \langle \texttt{make-cont}\; n_0\; n_1\; n_2 \rangle \mid \langle \texttt{literal}\; n \rangle \\
& \mid & \langle \texttt{closure}\; n \rangle \mid \langle \texttt{global}\; n \rangle \mid \langle \texttt{local}\; n_0\; n_1 \rangle \mid \langle \texttt{set-global!}\; n \rangle \\
& \mid & \langle \texttt{set-local!}\; n_0\; n_1 \rangle \mid \langle \texttt{push} \rangle \mid \langle \texttt{make-env}\; n \rangle \mid \langle \texttt{make-rest-list}\; n \rangle \\
& \mid & \langle \texttt{unspecified} \rangle \mid \langle \texttt{jump}\; n_0\; n_1 \rangle \mid \langle \texttt{jumpf}\; n_0\; n_1 \rangle \\
& \mid & \langle \texttt{checkargs=}\; n \rangle \mid \langle \texttt{checkargs>=}\; n \rangle \mid \langle i \rangle
\end{array}
$$

We no longer need the categories of blocks and of r-instructions, and m includes all kinds of instructions. Furthermore, comparing two uses of w, FBC code sequences are simpler than ABC instruction lists. Note that, here in FBC, every member of w is a sequence, of which each element is either a token representing the name of a byte code operation or a number. In the VLISP implementation, these numbers are all small enough unsigned integers to be stored in a single byte (i.e., less than 256).

4.2 The Augmented Flattened Byte Code Language

The purpose of the Augmented Flattened Byte Code Language AFBC is similar to that of the ABC, namely to extend the programming language to contain terms that can be used to describe the states of an FBC state machine, and to define its possible transitions.

As for the unflattened languages, all of the categories (templates, blocks etc.) and productions given for programming language (FBC) are included for the augmented language (AFBC), and the extensions of the BNF for the augmented language do not feed back into the recursive construction of the old categories, so their extensions in the AFBC are exactly the same as in the FBC.

The AFBC tokens not in FBC are the same as those of ABC not in TBC. The AFBC syntactic categories not in FBC correspond exactly to those of ABC not in TBC; here they are called (AFBC) *values*, (AFBC) *argument stacks*, etc. They use the same variable conventions as in ABC. The productions for the augmented categories v, a, u, and s are the same as in the ABC, but continuations are slightly different, the relevant BNF production is:

$$k \quad ::= \quad \text{HALT} \ | \ \langle \text{CONT}\ t\ n\ a\ u\ k \rangle$$

4.3 States and Actions

FBC states differ from TBC states in that they contain a template belonging to the FBC language rather than the TBC language. In addition, instead of having a "code register" they have a numerical "program counter." It is interpreted as an offset into the code sequence of the template.

Officially, the states of an FBC state machine are the sequences of the form

$$\langle t,\ n,\ v,\ a,\ u,\ k,\ s \rangle.$$

The components of a state are called, in order, its *template*, *offset*, *value*, *argument stack*, *environment*, *continuation*, and *store*, and we may informally speak of them as being held in registers. We will use the variable w to refer to the code sequence (or just the *code*) of a state, i.e., $w = t(1)$. Note that, with our conventions, $w(n)$ is the first element of $w \dagger n$, if $0 \leq n < \#w$. The *halt states* are the states such that $n = \#w$ (so $w \dagger n = \langle\rangle$).

Again, we present the *actions* relation for FBC state machines as the union of subfunctions called *(action) rules*, or FBC or AFBC rules to distinguish from TBC rules. The various rules are functions from pairwise disjoint subsets of *states* × *inputs* into *states* × *outputs*. The same distinction between *pure* rules and *port* rules is made as with the TBC rules, but, again, only pure rules will be specified at this level.

Each FBC machine also has an associated function *globals*, which still must be a function from some finite set of identifiers into locations.

We need another auxiliary function, *compute-offset*, which takes two integers and returns an integer representing an offset they are together coding. Keeping the size of a byte secret doesn't seem worth the trouble, so for $0 \leq n_0, n_1$,

$$\mathit{compute\text{-}offset}(n_0, n_1) = (256 * n_0) + n_1.$$

We will usually write $n_0 \oplus n_1$ for $\mathit{compute\text{-}offset}(n_0, n_1)$.

Note that, if one is very careful, three auxiliary functions, *mrl-store*, *mrl-value*, and *app-stack*, that were defined for TBC must be changed slightly for the FBC, because FBC values are slightly different. As the new definitions would look exactly the same, they are omitted, and we will use the same names for the FBC versions of these functions without danger of confusion.

The presentation of the pure rules for the FBC will use the same conventions as were used for the TBC, except that here n stands for the offset of the ingoing state, n' for the offset of the state produced, and w for $t(1)$, the code sequence of the template of the ingoing state.

4.3.1 Return-like Rules

Rule 1: **Return-Halt**
Domain conditions:

$w \dagger n = \langle \texttt{return} \rangle ^\frown w_1$

$k = \text{HALT}$

Changes:

$n' = \#w$

Rule 2: **Return**

Domain conditions:

$w \dagger n = \langle \texttt{return} \rangle ^\frown w_1$

$k = \langle \text{CONT } t_1 \; n_1 \; a_1 \; u_1 \; k_1 \rangle$

Changes:

$t' = t_1$

$n' = n_1$

$a' = a_1$

$u' = u_1$

$k' = k_1$

Rule 3: **Call**

Domain conditions:

$w \dagger n = \langle \texttt{call} \; \#a \rangle ^\frown w_1$

$v = \langle \text{CLOSURE } t_1 \; u_1 \; l_1 \rangle$

Changes:

$t' = t_1$

$n' = 0$

$u' = u_1$

Rule 4: **Escape-Halt**

Domain conditions:

$w \dagger n = \langle \texttt{call} \; 1 \rangle ^\frown w_1$

$v = \langle \text{ESCAPE HALT } l \rangle$

Changes:

$n' = \#w$

Rule 5: **Escape**
Domain conditions:

$w \dagger n = \langle \texttt{call}\ 1 \rangle {}^\frown w_1$
$v = \langle \text{ESCAPE}\ \langle \text{CONT}\ t_1\ n_1\ a_1\ u_1\ k_1 \rangle\ l \rangle$
$a = \langle v_1 \rangle$

Changes:

$t' = t_1$
$n' = n_1$
$v' = v_1$
$a' = a_1$
$u' = u_1$
$k' = k_1$

4.3.2 Jump-like Rules

Rule 6: **Jumpf/True**
Domain conditions:

$w \dagger n = \langle \texttt{jumpf}\ n_1\ n_2 \rangle {}^\frown w_1$
$v \neq false$

Changes:

$n' = n + 3$

Rule 7: **Jumpf/False**
Domain conditions:

$w \dagger n = \langle \texttt{jumpf}\ n_1\ n_2 \rangle {}^\frown w_1$
$v = false$

Changes:

$n' = n + 3 + (n_1 \oplus n_2)$

Rule 8: **Jump**
Domain conditions:

$w \dagger n = \langle \texttt{jump}\ n_1\ n_2 \rangle {}^\frown w_1$

Changes:

$n' = n + 3 + (n_1 \oplus n_2)$

4.3.3 Other Basic Rules

Rule 9: **Make Continuation**
Domain conditions:

$w \dagger n = \langle \texttt{make-cont}\ n_1\ n_2\ \#a \rangle ^\frown w_1$

Changes:

$n' = n + 4$

$a' = \langle \rangle$

$k' = \langle \text{CONT}\ t\ (n + 4 + (n_1 \oplus n_2))\ a\ u\ k \rangle$

Rule 10: **Literal**
Domain conditions:

$w \dagger n = \langle \texttt{literal}\ n_1 \rangle ^\frown w_1$

$e(n_1) = \langle \texttt{constant}\ c \rangle$

Changes:

$n' = n + 2$

$v' = c$

Rule 11: **Closure**
Domain conditions:

$w \dagger n = \langle \texttt{closure}\ n_1 \rangle ^\frown w_1$

$e(n_1) = \langle \texttt{template}\ b_2\ e_2 \rangle$

Changes:

$n' = n + 2$

$v' = \langle \text{CLOSURE}\ e(n_1)\ u\ \#s \rangle$

$s' = s ^\frown \langle \text{NOT-SPECIFIED} \rangle$

Rule 12: **Global**
Domain conditions:

$w \dagger n = \langle \texttt{global}\ n_1 \rangle ^\frown w_1$

$e(n_1) = \langle \texttt{global-variable}\ i \rangle$

$l = \mathit{globals}(i)$

$v_1 = s(l)$

$v_1 \neq \text{UNDEFINED}$

Changes:

$n' = n + 2$

$v' = v_1$

Rule 13: **Set Global**
Domain conditions:
$w \dagger n = \langle \texttt{set-global!}\ n_1 \rangle ^\frown w_1$
$e(n_1) = \langle \texttt{global-variable}\ i \rangle$
$l = \mathit{globals}(i)$
$l \leq \# s$
Changes:
$n' = n + 2$
$v' = \text{NOT-SPECIFIED}$
$s' = s + \{l \mapsto v\}$

Rule 14: **Local**
Domain conditions:
$w \dagger n = \langle \texttt{local}\ n_1\ n_2 \rangle ^\frown w_1$
$l = \mathit{env\text{-}reference}(u,\ n_1,\ n_2)$
$v_1 = s(l)$
$v_1 \neq \text{UNDEFINED}$
Changes:
$n' = n + 3$
$v' = v_1$

Rule 15: **Set Local**
Domain conditions:
$w \dagger n = \langle \texttt{set-local!}\ n_1\ n_2 \rangle ^\frown w_1$
$l = \mathit{env\text{-}reference}(u,\ n_1,\ n_2)$
$l \leq \# s$
Changes:
$n' = n + 3$
$v' = \text{NOT-SPECIFIED}$
$s' = s + \{l \mapsto v\}$

Rule 16: **Push**
Domain Conditions:
$w \dagger n = \langle \texttt{push} \rangle ^\frown w_1$
Changes:
$n' = n + 1$
$a' = v :: a$

Rule 17: **Make Environment**
Domain Conditions:
$w\dagger n = \langle \texttt{make-env}\ \#a\rangle^\frown w_1$
Changes:
$n' = n + 2$
$a' = \langle\rangle$
$u' = \textit{add-layer}(u,\ \#s,\ \#a)$
$s' = s^\frown a$

Rule 18: **Make Rest List**
Domain Conditions:
$w\dagger n = \langle \texttt{make-rest-list}\ n_1\rangle^\frown w_1$
$n_1 + n_2 = \#a$
Changes:
$n' = n + 2$
$v' = \textit{mrl-value}\,(n_2,\ \textsc{nil},\ a,\ s)$
$a' = a \dagger n_2$
$s' = \textit{mrl-store}\,(n_2,\ \textsc{nil},\ a,\ s)$

Rule 19: **Unspecified**
Domain Conditions:
$w\dagger n = \langle \texttt{unspecified}\rangle^\frown w_1$
Changes:
$n' = n + 1$
$v' = \textsc{not-specified}$

Rule 20: **Check Args =**
Domain Conditions:
$w\dagger n = \langle \texttt{checkargs=}\ n_1\rangle^\frown w_1$
$\#a = n_1$
Changes:
$n' = n + 2$

Rule 21: **Check Args** >=
Domain Conditions:
$w\dagger n = \langle \texttt{checkargs=}\ n_1\rangle^\frown w_1$
$\#a \geq n_1$
Changes:
$n' = n + 2$

4.3.4 Primitive Operation Rules

Rule 22: **Primitive CWCC**

Domain Conditions:

$w \dagger n = \langle \texttt{\%\%cwcc} \rangle ^\frown w_1$

$a = \langle v_1 \rangle$

$v_1 = \langle \text{CLOSURE } t_1 \ u_1 \ l_1 \rangle$

Changes:

$t' = t_1$

$n' = 0$

$v' = v_1$

$a' = \langle \langle \text{ESCAPE } k \ \#s \rangle \rangle$

$u' = u_1$

$s' = s ^\frown \langle \text{NOT-SPECIFIED} \rangle$

Rule 23: **Primitive CWCC-Escape**

Domain Conditions:

$w \dagger n = \langle \texttt{\%\%cwcc} \rangle ^\frown w_1$

$a = \langle \langle \text{ESCAPE } \langle \text{CONT } t_1 \ n_1 \ a_1 \ u_1 \ k_1 \rangle \ l_1 \rangle \rangle$

Changes:

$t' = t_1$

$n' = n_1$

$v' = \langle \text{ESCAPE } k \ \#s \rangle$

$a' = a_1$

$u' = u_1$

$k' = k_1$

$s' = s ^\frown \langle \text{NOT-SPECIFIED} \rangle$

Rule 24: **Primitive CWCC-Escape-Halt**

Domain Conditions:

$w \dagger n = \langle \texttt{\%\%cwcc} \rangle ^\frown w_1$

$a = \langle \langle \text{ESCAPE HALT } l \rangle \rangle$

Changes:

$n' = \#w$

$v' = \langle \text{ESCAPE } k \ \#s \rangle$

Rule 25: **Primitive Cons**
Domain Conditions:
$w \dagger n = \langle \texttt{\%\%cons} \rangle \frown w_1$
$a = \langle v_1 \ v_2 \rangle \frown a_1$
Changes:
$n' = n + 1$
$v' = \langle \textsc{mutable-pair} \ \#s \ (1 + \#s) \rangle$
$a' = \langle \rangle$
$s' = s \frown \langle v_2 \ v_1 \rangle$

Rule 26: **Primitive Car-Immutable Pair**
Domain Conditions:
$w \dagger n = \langle \texttt{\%\%car} \rangle \frown w_1$
$a = \langle \texttt{immutable-pair} \ c_1 \ c_2 \rangle :: a_1$
Changes:
$n' = n + 1$
$v' = c_1$
$a' = \langle \rangle$

Rule 27: **Primitive Car-Mutable Pair**
Domain Conditions:
$w \dagger n = \langle \texttt{\%\%car} \rangle \frown w_1$
$a = \langle \textsc{mutable-pair} \ l_1 \ l_2 \rangle :: a_1$
$v_1 = s(l_1)$
Changes:
$n' = n + 1$
$v' = v_1$
$a' = \langle \rangle$

Rule 28: **Primitive Set-Car!**
Domain Conditions:
$w \dagger n = \langle \texttt{\%\%set-car!} \rangle \frown w_1$
$a = \langle v_1 \ \langle \textsc{mutable-pair} \ l_1 \ l_2 \rangle \rangle \frown a_1$
$l_1 < \#s$
Changes:
$n' = n + 1$
$v' = \textsc{not-specified}$
$a' = \langle \rangle$
$s' = s + \{ l_1 \mapsto v_1 \}$

Rule 29: **Primitive Apply-Closure**
Domain Conditions:

$w \dagger n = \langle \texttt{\%\%apply} \rangle ^\frown w_1$

$a = \langle v_1 \rangle ^\frown a_1 ^\frown \langle v_2 \rangle$

$v_2 = \langle \text{CLOSURE } t_1\ u_1\ l_1 \rangle$

$t_1 = \langle \texttt{template}\ b_2\ e_2 \rangle$

$a_2 = \textit{app-stack}(v_1,\ a_1,\ s)$

Changes:

$t' = t_1$

$n' = 0$

$v' = v_2$

$a' = a_2$

$u' = u_1$

Rule 30: **Primitive Apply-Escape**
Domain Conditions:

$w \dagger n = \langle \texttt{\%\%apply} \rangle ^\frown w_1$

$a = \langle v_1 \rangle ^\frown a_1 ^\frown \langle v_2 \rangle$

$v_2 = \langle \text{ESCAPE } k_1\ l_1 \rangle$

$k_1 = \langle \text{CONT } t_2\ n_2\ a_2\ u_2\ k_2 \rangle$

$\langle v_3 \rangle = \textit{app-stack}(v_1,\ a_1,\ s)$

Changes:

$t' = t_2$

$n' = n_2$

$v' = v_3$

$a' = a_2$

$u' = u_2$

$k' = k_2$

Rule 31: **Primitive Apply-Escape-Halt**
Domain Conditions:

$w \dagger n = \langle \texttt{\%\%apply} \rangle ^\frown w_1$

$a = \langle v_1 \rangle ^\frown a_1 ^\frown \langle v_2 \rangle$

$v_2 = \langle \text{CLOSURE } t_1\ u_1\ l_1 \rangle$

$v_2 = \langle \text{ESCAPE HALT } l_1 \rangle$

Changes:

$n' = \#w$

$v' = v_2$

Rule 32: **Primitive Eqv**
Domain Conditions:
$w \dagger n = \langle \texttt{\%\%eqv} \rangle ^\frown w_1$
$a = \langle v_1\ v_2 \rangle ^\frown a_1$
Changes:
$n' = n + 1$
$v' = ((v_1 = v_2) \to true, false)$
$a' = \langle \rangle$

Rule 33: **Primitive Add**
Domain Conditions:
$w \dagger n = \langle \texttt{\%\%add} \rangle ^\frown w_1$
$m = n\text{-}ary\text{-}sum\,(a)$
Changes:
$n' = n + 1$
$v' = m$
$a' = \langle \rangle$

5 The Flattener Algorithm

In this section, we present the applicative Scheme code that implements the flattener. In addition, we also consider it as a (perhaps slightly unusual) way to present a mathematical function.

5.1 The Flattener Proper

```
(define (flatten-template tem)
  `(template
    ,(raw-code (flatten-code (cadr tem) 'closed))
    ,(flatten-table (caddr tem))))

(define (flatten-table table)
  (map (lambda (entry)
         (if (tbc-template? entry)
             (flatten-template entry)
             entry))
       table))

(define (flatten-code code category)
  (if (null? code)
      empty-code-sequence
      (let ((instr (car code))
            (after-code (cdr code)))
        (case (operator instr)
          ((make-cont)
           (flatten-make-cont (first-operand instr)
                              (second-operand instr)
                              after-code
                              category))
          ((unless-false)
           (if (and (eq? category 'closed)
                    (null? after-code))
               (flatten-if-not-false (first-operand instr)
                                     (second-operand instr))
               (flatten-unless-false (first-operand instr)
                                     (second-operand instr)
                                     after-code
                                     category)))
          (else
           (flatten-normal-instruction instr
                                       after-code
                                       category))))))
```

```
(define (flatten-normal-instruction instr after category)
  (let ((after-code-sequence (flatten-code after category)))
    (prepend-instruction instr after-code-sequence)))

(define (flatten-make-cont
         saved-code nargs after-code
         category)
  (let ((after-code (flatten-code after-code 'closed))
        (saved-code (flatten-code saved-code category)))
    (add-offset+byte-instruction
     'make-cont
     (code-length after-code)
     nargs
     (adjoin-code-sequences after-code saved-code))))

(define (flatten-if-not-false true-branch false-branch)
  (let ((con (flatten-code true-branch 'closed))
        (alt (flatten-code false-branch 'closed)))
    (add-offset-instruction
     'jump-if-false
     (code-length con)
     (adjoin-code-sequences con alt))))

(define (flatten-unless-false
         true-code false-code after-code
         category)
  (let ((con (flatten-code true-code 'open))
        (alt (flatten-code false-code 'open))
        (after-code (flatten-code after-code category)))
    (add-offset-instruction
     'jump-if-false
     (+ 3                              ; length of jump instr
        (code-length con))
     (adjoin-code-sequences
      con
      (add-offset-instruction
       'jump (code-length alt)
       (adjoin-code-sequences alt after-code))))))
```

5.2 Auxiliary Procedures

```
(define (first-operand instr) (cadr instr))
(define (second-operand instr) (caddr instr))
(define (operator instr) (car instr))
```

```
(define (add-offset-instruction name offset after-code)
  (prepend-instruction
  `(,name ,@(expand-offset offset))
   after-code))

(define (add-offset+byte-instruction name offset
                                     byte after-code)
  (prepend-instruction
  `(,name ,@(expand-offset offset) ,byte)
   after-code))

(define byte-limit 256)

(define (tbc-template? entry)
  (and (pair? entry)
       (eq? (car entry) 'template)))

(define (expand-offset offset)
  (list (quotient offset byte-limit)
        (remainder offset byte-limit)))
```

Correctness depends on the fact that `expand-offset` and $\oplus$ are effectively inverses:

Lemma 2 Expand-offset/$\oplus$.
Let $\langle n_0, n_1 \rangle$ *be the result of applying* `expand-offset` *to* n. *Then* $n_0 \oplus n_1 = n$.

5.3 The Code Sequence Data Type

The following five identifiers implement the data type of code sequences.

```
(define empty-code-sequence 'declared)
(define prepend-instruction 'declared)
(define adjoin-code-sequences 'declared)
(define raw-code 'declared)
(define code-length 'declared)

(let ((code-sequence-marker (list 'code-sequence-marker)))
  (let ((make-code-sequence
         (lambda (raw-code len)
           (list code-sequence-marker raw-code len)))
        (code-sequence?
         (lambda (cs)
           (and (pair? cs)
```

```
              (eq? (car cs) code-sequence-marker)))))
(set! empty-code-sequence (make-code-sequence '() 0))
(set!
 prepend-instruction
 (lambda (instr after-code)
   (make-code-sequence
    (append instr (raw-code after-code))
    (+ (length instr) (code-length after-code)))))
(set!
 adjoin-code-sequences
 (lambda (first second)
   (make-code-sequence
    (append (raw-code first)
            (raw-code second))
    (+ (code-length first)
       (code-length second)))))
(set!
 raw-code
 (lambda (code-sequence)
   (if (code-sequence? code-sequence)
       (cadr code-sequence)
       (compiler-error
        "raw-code -- bad code sequence"
        code-sequence))))
(set!
 code-length
 (lambda (code-sequence)
   (if (code-sequence? code-sequence)
       (caddr code-sequence)
       (compiler-error
        "code-length -- bad code sequence"
        code-sequence))))))
```

If s is a code sequence, the length of the raw code contained in s is equal to the code-length of s. This holds true of the empty code sequence. Moreover, it is preserved under the operations of prepending an instruction and adjoining sequences, because the length of the result of appending a number of lists is equal to the sum of their lengths.

6 Establishing Correspondence of Code

We shall eventually give the notion of correspondence between states in terms of a preliminary "code correspondence" relation $\simeq$. This is really a four-place relation on a TBC instruction list, a TBC table, a FBC code sequence, and a FBC table. We think of it as a binary correspondence between the pair of its first two arguments and the pair of its last two arguments, written, for instance,

$$(w, e) \;\simeq\; (w_F, e_F).$$

It is defined as the least relation satisfying the conditions summarized in Table 2. One fine point in the definition concerns expressions of the form $w\dagger n$: when we make an atomic assertion about $w\dagger n$, we are implicitly asserting that it is well defined, so that $n \leq \#w$.

We will repeatedly use a fact about sequences:

Lemma 3 drop/adjoin. *When $n \leq \#w_0$, $(w_0\dagger n)^\frown w_1 = (w_0{}^\frown w_1)\dagger n$.*

For *closed* instruction lists, adding code to the end of a flattened version does not destroy the $\simeq$ relation:

Lemma 4 $\simeq$/adjoin. *If $(b_0, e) \;\simeq\; (w_0, e^F)$, then for any w_1:*

$$(b_0, e) \;\simeq\; (w_0{}^\frown w_1, e^F).$$

Proof. We will assume that w_0 is not of the form $\langle \texttt{jump}\, n_0\; n_1\rangle^\frown w_2$, and thus that $(b_0, e) \;\simeq\; (w_0{}^\frown w_1, e^F)$ is not true in virtue of clause 6. For otherwise, $(b_0, e) \;\simeq\; (w_2\dagger(n_0 \oplus n_1), e^F)$, and we may apply the lemma[1] to infer that

$$(b_0, e) \;\simeq\; (w_2\dagger(n_0 \oplus n_1)^\frown w_1, e^F).$$

Since $w_2\dagger(n_0 \oplus n_1)^\frown w_1 = w_2{}^\frown w_1\dagger(n_0 \oplus n_1)$, we may apply clause 6 to infer:

$$(b_0, e) \;\simeq\; (\langle \texttt{jump}\, n_0\; n_1\rangle^\frown(w_2{}^\frown w_1), e^F).$$

By the associativity of $^\frown$, the desired conclusion holds in this case also.

The proof is by induction mirroring the inductive definition of $\simeq$.

1. Suppose $b_0 = \langle m\rangle = \langle\langle \texttt{return}\rangle\rangle$ or $\langle\langle \texttt{call}\; n\rangle\rangle$, and $w_0 = \langle m\rangle^\frown w$. By the associativity of $^\frown$, $(\langle m\rangle^\frown w)^\frown w_1 = \langle m\rangle^\frown(w^\frown w_1)$, which is also an instance of clause 1.

[1] Naturally, $w_2\dagger(n_0 \oplus n_1)$ might begin with a jump, but this may be repeated only a finite number of times.

1. For $m = \langle \texttt{return} \rangle$ or $\langle \texttt{call}\ n \rangle$, $(\langle m \rangle, e) \ \simeq \ (m^\frown w_F, e_F)$.

2. For atomic z, $(z :: m^*, e) \ \simeq \ (z^\frown w_F, e_F)$ if, first, either $m^* = w_F = \langle\rangle$ or $(m^*, e) \ \simeq \ (w_F, e_F)$, and second, depending on the form of z:

   (a) if $z = \langle \texttt{literal}\ n \rangle$, $\langle \texttt{global}\ n \rangle$, or $\langle \texttt{set-global!}\ n \rangle$, then $e(n) = e_F(n)$;

   (b) if $z = \langle \texttt{closure}\ n \rangle$, then:

      i. $e(n)$ is of the form $\langle \texttt{template}\ b_1\ e_1 \rangle$, and

      ii. $e_F(n)$ is of the form $\langle \texttt{template}\ w_2\ e_2 \rangle$, where

      iii. $(b_1, e_1) \ \simeq \ (w_2, e_2)$.

3. $(\langle \texttt{make-cont}\ w\ n \rangle :: b, e) \ \simeq \ (\langle \texttt{make-cont}\ n_0\ n_1\ n \rangle^\frown w_F, e_F)$ if:

   (a) $(b, e) \ \simeq \ (w_F, e_F)$; and

   (b) $(w, e) \ \simeq \ (w_F \dagger (n_0 \oplus n_1), e_F)$.

3′. $(\langle \texttt{make-cont}\ \langle\rangle\ n \rangle :: b, e) \ \simeq \ (\langle \texttt{make-cont}\ n_0\ n_1\ n \rangle^\frown w_F, e_F)$ if:

   (a) $(b, e) \ \simeq \ (w_F, e_F)$; and

   (b) $\# w_F = n_0 \oplus n_1$.

4. $(\langle\langle \texttt{unless-false}\ b_1\ b_2 \rangle\rangle, e) \ \simeq \ (\langle \texttt{jumpf}\ n_0\ n_1 \rangle^\frown w_F, e_F)$ if:

   (a) $(b_1, e) \ \simeq \ (w_F, e_F)$; and

   (b) $(b_2, e) \ \simeq \ (w_F \dagger (n_0 \oplus n_1), e_F)$.

5. $(\langle \texttt{unless-false}\ y_1\ y_2 \rangle :: w, e) \ \simeq \ (\langle \texttt{jumpf}\ n_0\ n_1 \rangle^\frown w_F, e_F)$ if:

   (a) $(y_1{}^\smile w, e) \ \simeq \ (w_F, e_F)$;

   (b) $(y_2{}^\smile w, e) \ \simeq \ (w_F \dagger (n_0 \oplus n_1), e_F)$.

6. $(w, e) \ \simeq \ (\langle \texttt{jump}\ n_0\ n_1 \rangle^\frown w_F, e_F)$ if $(w, e) \ \simeq \ (w_F \dagger (n_0 \oplus n_1), e_F)$.

Table 2: Recursive Conditions for $\simeq$.

2. Suppose that $b_0 = z :: b$, $w_0 = z^\frown w$. (The syntax ensures that m^* really is of the form b.) Assume inductively that $(b, e) \simeq (w^\frown w_1, e_F)$. Then we simply apply clause 2 to $w^\frown w_1$. The subclauses dealing with e and e_F are unaffected.

3. If $b_0 = \langle \texttt{make-cont}\ b_1\ n \rangle :: b$, and $w_0 = \langle \texttt{make-cont}\ n_0\ n_1\ n_2 \rangle^\frown w$: Assume inductively:
$$(b, e) \simeq (w^\frown w_1, e_F)$$
and
$$(b_1, e) \simeq (w \dagger (n_0 \oplus n_1)^\frown w_1, e_F).$$
By the drop/adjoin lemma, we may apply clause 3 with $w^\frown w_1$ in place of w.

3′. A closed instruction sequence b_0 is not of this form.

4. Similar to 3.

5. Similar to 3, using the fact that $y_i{}^\smile b$ forms a closed instruction list b_2.

6. Similar to 3. QED.

On the other hand, for *open* instruction lists, adding code to the end of a flattened version corresponds to $^\smile$:

Lemma 5 **$\simeq$/open-adjoin.** *If* $(y, e) \simeq (w_0, e^F)$ *and* $(w, e) \simeq (w_1, e^F)$, *then*
$$(y^\smile w, e) \simeq (w_0{}^\frown w_1, e^F).$$

Proof. As in the previous proof, we may assume that w_0 is not of the form $\langle \texttt{jump}\ n_0\ n_1 \rangle^\frown w_2$, and thus that the correspondence does not hold in virtue of clause 6.

The proof is by an induction on y. We will arrange the cases to correspond to the clauses in the inductive definition of $\simeq$, although the syntax of y entails that only the clauses 2, 3, 5, and 6 are relevant. We divide clause 2 into two subcases, depending whether $m^* = \langle\rangle$ or $m^* = y_0$. N. B. In this proof, subscripted variables w_i range over FBC instruction lists. The unsubscripted w ranges over TBC instruction lists.

2a. Assume $y = \langle z \rangle$. Then $w_0 = z$, so clause 2 (applied to $z :: w$) guarantees the conclusion. The subclauses hold for $z :: w$ for the same reason they hold for $\langle z \rangle$.

2b. Assume $y = z :: y_0$. Then $w_0 = z^\frown w_2$, and by the IH,

$$(y_0{}^\smile w, e) \;\simeq\; (w_2{}^\frown w_1, e^F).$$

But $(z :: y_0)^\smile w = z :: (y_0{}^\smile w)$, so we may again apply clause 2.

3. Assume $y = \langle\texttt{make-cont}\ y_0\ n\rangle :: b$; so

$$y^\smile w = \langle\texttt{make-cont}\ y_0{}^\smile w\ n\rangle :: b.$$

Moreover, $w_0 = \langle\texttt{make-cont}\ n_0\ n_1\ n_2\rangle^\frown w_2$. By the $\simeq$/adjoin lemma, $(b, e) \;\simeq\; (w_2, e^F)$ entails that $(b, e) \;\simeq\; (w_2{}^\frown w_1, e^F)$. By the IH and the drop/adjoin lemma, $(y_0, e) \;\simeq\; (w_2 \dagger (n_0 \oplus n_1), e^F)$ implies

$$(y_0{}^\smile w, e) \;\simeq\; ((w_2{}^\frown w_1) \dagger (n_0 \oplus n_1), e^F).$$

3′. Assume $y = \langle\texttt{make-cont}\ \langle\rangle\ n\rangle :: b$; so

$$y^\smile w = \langle\texttt{make-cont}\ w\ n\rangle :: b.$$

So $w_0 = \langle\texttt{make-cont}\ n_0\ n_1\ n_2\rangle^\frown w_2$. We will show that $y^\smile w$ and $w_0{}^\frown w_1$ satisfy Clause 3 rather than Clause 3′.

As in the preceeding case, by the $\simeq$/adjoin lemma, $(b, e) \;\simeq\; (w_2, e^F)$ entails that $(b, e) \;\simeq\; (w_2{}^\frown w_1, e^F)$. So Clause 3(a) is satisfied.

Because $(y, e) \;\simeq\; (w_0, e^F)$, we may apply Clause 3′(b) to conclude that $n_0 \oplus n_1 = \# w_2$. Hence $w_2{}^\frown w_1 \dagger (n_0 \oplus n_1) = w_1$, so Clause 3(b) is also satisfied.

5. Assume $y = \langle\texttt{unless-false}\ y_1\ y_2\rangle :: y_3$. So $w_0 = \langle\texttt{jumpf}\ n_0\ n_1\rangle^\frown w_2$. Apply the IH to $y_1{}^\smile y_3$ and $y_2{}^\smile y_3$, using the drop/adjoin lemma.

6. As before. QED.

The flattener is intended to apply only to templates $\langle\texttt{template}\ b\ e\rangle$ that have been generated by the tabulator. These have the property that whenever we encounter an instruction $\langle\texttt{closure}\ n\rangle$, then $e(n)$ is in fact a template. Conversely, when we encounter an instruction $\langle\texttt{literal}\ n\rangle$, $\langle\texttt{global}\ n\rangle$, or $\langle\texttt{set-global!}\ n\rangle$, then $e(n)$ is *not* a template.

Definition 6 *An instruction m* occurs in *an instruction sequence w if:*

1. $w = \langle m\rangle$*;*

2. $w = m_1 :: w_1$ *and either* $m = m_1$ *or* m *occurs in* w_1*;*

3. $w = \langle\langle$`unless-false` $b_1\ b_2\rangle\rangle$ *or* $w = \langle$`make-cont` $b_1\ n\rangle :: b_2$*, and* m *occurs in* b_1 *or* b_2*;*

4. $w = \langle$`make-cont` $y_1\ n\rangle :: b$ *and* m *occurs in* y_1 *or* b*.*

If z *is a* TBC *neutral instruction, then* z respects the template table e *if:*

1. if $z = \langle$`closure` $n\rangle$, then $e(n)$ is a template $\langle$`template` $b_1\ e_1\rangle$, and moreover b_1 (recursively) respects the template table e_1;

2. if $z = \langle$`literal` $n\rangle$, then $e(n)$ is a literal of the form $\langle$`constant` $c\rangle$; and

3. if $z = \langle$`global` $n\rangle$ or $\langle$`set-global!` $n\rangle$, then $e(n)$ is a global of the form $\langle$`global` $i\rangle$.

A TBC *closed or open instruction sequence* w respects the template table e *if every* z *that occurs in* w *respects* e*.*

If t *is a template* $\langle$`template` $b\ e\rangle$*, we will say that* t *respects its table, or that* t *is* self-respecting*, if* b *respects the template table* e*.*

If t *is a template* $\langle$`template` $b\ e\rangle$*, we will say that* t *respects its table, or that* t *is* self-respecting*, if* b *respects the template table* e*.*

In the treatment of the tabulator, we have proved a lemma that states that the output of the tabulator always has this property. That is, if the result of the tabulator is a TBC template $\langle$`template` $b\ e\rangle$, then b respects the template table e. This provides the justification for proving the correctness of the flattener only for input templates with this property.

We will use $F(w)$ to abbreviate the result of applying `flatten-code` to w and *closed*, if w is of the form b, and for the result of applying `flatten-code` to w and *open*, if w is of the form y. Similarly, we will use $F(e)$ as an abbreviation for the result of applying `flatten-table` to e, and we will use $F(t)$ to mean the result of applying `flatten-template` to t. Observe from the code that $F(\langle$`template` $b\ e\rangle)$ is $\langle$`template` $F(b)\ F(e)\rangle$.

Lemma 7 No initial jumps. $F(w)$ *is not of the form* $\langle$`jump` $n_0\ n_1\rangle^\frown w_1$*.*

Proof. The flattener code ensures:

1. If w begins with a `make-cont`, then the first instruction of $F(w)$ is a `make-cont`;

2. If w begins with an `unless-false`, then the first instruction of $F(w)$ is a `jumpf`;

3. Otherwise, $F(w)$ begins with the same instruction as w. QED.

Theorem 8 Flattener establishes $\simeq$.
Suppose that w respects the template table e.

A. $(w, e) \ \simeq \ (F(w), F(e))$*;*

B. If n occurs in w and $e(n)$ is of the form $\langle$`template` b_1 $e_1\rangle$*, then*

$$(b_1, e_1) \ \simeq \ (F(w_1), F(e_1)).$$

Proof. The proof is by simultaneous induction on the structure of w and the depth (in nested templates) of e.

A. $(w, e) \ \simeq \ (F(w), F(e))$: We assume inductively that:

1. Part B holds true for e;

2. If w is of the form $\langle\langle$`unless-false` b_1 $b_2\rangle\rangle$, then Part A holds true for b_1 and b_2 (together with the same e, naturally);

3. If w is of the form $m :: w_0$, then Part A holds true for w_0, and moreover:

   (a) if m is of the form $\langle$`make-cont` w_1 $n\rangle$, then Part A holds true for w_1;

   (b) If m is of the form $\langle$`unless-false` w_1 $w_2\rangle$, then Part A holds true for w_1 and w_2.

The proof is by cases on the grammar of w. However, we will list them in the order corresponding to the first five clauses of the definition of $\simeq$. By Lemma 7, the initial instruction of $F(w_0)$ is never of the form `jump`. Hence, a correspondence involving an expression of this form is never true in virtue of clause 6.

1. $w = m = \langle\langle$`return`$\rangle\rangle$ or $\langle\langle$`call` $n\rangle\rangle$. Then, by the algorithm, $F(w) = m$, satisfying clause 1.

2. $w = \langle z\rangle$, for atomic z. Then $F(w) = z^\frown\langle\rangle = z$, and we may apply clause 2, noting:

(a) Suppose $z = \langle \texttt{literal}\ n \rangle$: Then, because w respects e, $e(n)$ is of the form $\langle \texttt{constant}\ c \rangle$, and the code of `flatten-table` ensures that $e(n) = F(e)(n)$.

(b) Suppose $z = \langle \texttt{global}\ n \rangle$ or $\langle \texttt{set-global!}\ n \rangle$: Then, because w respects e, $e(n)$ is of the form $\langle \texttt{global}\ i \rangle$, and the code of `flatten-table` ensures that $e(n) = F(e)(n)$.

(c) Suppose $z = \langle \texttt{closure}\ n \rangle$: Then, because n occurs in w and w respects e, $e(n)$ is of the form $\langle \texttt{template}\ b_1\ e_1 \rangle$. Hence,

$$F(e)(n) = \langle \texttt{template}\ F(b_1)\ F(e_1) \rangle.$$

So part 1 of the IH entails that clause 2b of $\simeq$ is satisfied.

2′. $w = z :: w_0$, for atomic z. Then $F(w) = z ^\frown F(w_0)$. By the IH and the same observations we made in the previous case, we may again apply clause 2.

3. $w = \langle \texttt{make-cont}\ w_1\ n \rangle :: b_0$. Then

$$F(w) = \langle \texttt{make-cont}\ n_0\ n_1\ n \rangle ^\frown F(b_0) ^\frown F(w_1),$$

where $\langle n_0, n_1 \rangle = \texttt{expand-offset}(\#F(b_0))$.

(a) IH 3 ensures $(b_0, e) \simeq (F(b_0), F(e))$, and $\simeq$/adjoin allows us to infer $(b_0, e) \simeq (F(b_0) ^\frown F(w_1), F(e))$. Hence clause 3a in $\simeq$ is satisfied.

(b) Using the expand-offset/$\oplus$ lemma, $(F(b_0) ^\frown F(w_1))\dagger(n_0 \oplus n_1) = F(w_1)$. Hence, by IH part 3a, clause 3b in $\simeq$ is satisfied.

3′. $w = \langle \texttt{make-cont}\ \langle\rangle\ n \rangle :: b_0$. Then

$$F(w) = \langle \texttt{make-cont}\ n_0\ n_1\ n \rangle ^\frown F(b_0) ^\frown \langle\rangle,$$

where $\langle n_0, n_1 \rangle = \texttt{expand-offset}(\#F(b_0))$.

(a) IH 3 ensures $(b_0, e) \simeq (F(b_0), F(e))$, and $\simeq$/adjoin allows us to infer $(b_0, e) \simeq (F(b_0) ^\frown F(w_1), F(e))$. Hence clause 3′(a) in $\simeq$ is satisfied.

(b) $n_0 \oplus n_1 = \#F(b_0)$ by Lemma expand-offset/$\oplus$.

4. $w = \langle\langle \texttt{unless-false}\; b_1\; b_2 \rangle\rangle$. Then

$$F(w) = \langle \texttt{jumpf}\; n_0\; n_1 \rangle ^\frown F(b_1) ^\frown F(b_2),$$

where $\langle n_0, n_1 \rangle = \texttt{expand-offset}(\#F(b_1))$.

   (a) IH part 2 gives $(b_1, e) \simeq (F(b_1), F(e))$. The $\simeq$/adjoin lemma entails that $(b_1, e) \simeq (F(b_1)^\frown F(b_2), F(e))$, satisfying clause 4a in $\simeq$.

   (b) Using the expand-offset/$\oplus$ lemma, $F(b_1)^\frown F(b_2)\dagger(n_0 \oplus n_1) = F(b_2)$, so IH part 2 ensures that clause 4b in $\simeq$ is satisfied.

5. $w = \langle \texttt{unless-false}\; y_1\; y_2 \rangle :: w_1$. Then

$$F(w) = \langle \texttt{jumpf}\; n_0\; n_1 \rangle ^\frown F(y_1) ^\frown \langle \texttt{jump}\; n_2\; n_3 \rangle ^\frown F(y_2) ^\frown F(w_1),$$

where $n_0 \oplus n_1 = 3 + \#F(y_1)$, and $n_2 \oplus n_3 = \#F(y_2)$. IH part 3 ensures that $(w_1, e) \simeq (F(w_1), F(e))$.

   (a) Clause 6 entails:

$$(w_1, e) \simeq (\langle \texttt{jump}\; n_2\; n_3 \rangle ^\frown F(y_2) ^\frown F(w_1), F(e)).$$

   IH part 3b ensures that $(y_1, e) \simeq (F(y_1), F(e))$, so by $\simeq$/open-adjoin,

$$(y_1 {}^\smile w_1, e) \simeq (F(y_1) ^\frown \langle \texttt{jump}\; n_2\; n_3 \rangle ^\frown F(y_2) ^\frown F(w_1), F(e)).$$

   (b) IH part 3b ensures that $(y_2, e) \simeq (F(y_2), F(e))$, so by $\simeq$/open-adjoin,

$$(y_2 {}^\smile w_1, e) \simeq (F(y_2) ^\frown F(w_1), F(e)).$$

   Since $n_0 \oplus n_1 = \#(F(y_1) ^\frown \langle \texttt{jump}\; n_2\; n_3 \rangle)$, we are done.

B. Here we assume inductively that Part A holds for all w_1, and all less deeply nested e_1, and that n occurs in w.

But if $e(n)$ is of the form $\langle \texttt{template}\; b_1\; e_1 \rangle$, then e_1 is less deeply nested. Thus, applying Part A to b_1 and e_1, $(b_1, e_1) \simeq (F(b_1), F(e_1))$, as desired. QED.

7 Preservation of State Correspondence

Based on the underlying "code correspondence" relation $\simeq$, we will define a binary "miscellaneous correspondence" relation $\sim$ between various kinds of syntactic objects of ABC on the left and ones of AFBC on the right, building up to a notion of correspondence between TBC states and FBC states that is preserved by state transitions.

We take advantage of the inessential fact that the two languages share some syntactic objects to simplify the definition. Thus some of the syntactic objects that are shared by the two languages are to be called *self corresponding*, namely, the HALT continuation; all environments; UNDEFINED; NOT-SPECIFIED; all constants; and all values beginning with MUTABLE-PAIR, STRING, or VECTOR. The relation $\sim$ is then defined recursively, as the least set of pairs whose left element is a member of one of the ABC classes t, v, a, u, k or s, or is an ABC state, and which satisfies the closure conditions given in Table 3. Note especially clauses (6) and (7), which reflect differences in how, and even in which registers, the two machines store "active" code. Also, case (4) applies to both argument stacks and stores.

The following easy lemma shortens the proof of the preservation theorem somewhat and justifies a looser way of thinking about computations of TBC and FBC state machines. One can talk about what they *do* (or *would do* under given circumstances), not just about what they *might* do.

Lemma 9 Determinacy of Pure Transitions.
If M is either a TBC machine or an FBC machine, and M is in state S, then there is at most one state S' such that M can proceed by one pure rule transition from S to S'.

Proof. The domain conditions for the different rules (of the same machine) are pairwise mutually incompatible, as can be seen by looking just at the required form of the code register, except for a few cases when one must also consider the value register or the argument register. Given a particular pure rule and a state, there is always at most one way of binding the variables of the domain equations so that they are satisfied. The next state is clearly determined by the rule, the ingoing state, and these bindings. QED.

Clearly the TBC rules are closely paralleled by the FBC rules – let us say that a TBC rule A *corresponds* to an FBC rule B if A and B have the same name, also that Closed Branch/True and Open Branch/True both *correspond* to Jumpf/True, and finally that Closed Branch/False and Open Branch/False

(1) $x \sim x$, if x is self corresponding

(2) $\langle \text{CLOSURE } t_1\ u\ l\rangle \sim \langle \text{CLOSURE } t_2\ u\ l\rangle$, if $t_1 \sim t_2$

(3) $\langle \text{ESCAPE } k_1\ l\rangle \sim \langle \text{ESCAPE } k_2\ l\rangle$, if $k_1 \sim k_2$

(4) $v_1^* \sim v_2^*$, if $\#v_1^* = \#v_2^*$ and, for every $n < \#v_1^*$, $v_1^*(n) \sim v_2^*(n)$

(5) $\langle \texttt{template}\ b\ e_1\rangle \sim \langle \texttt{template}\ w\ e_2\rangle$, if $(b,\ e_1) \simeq (w,\ e_2)$

(6) $\langle \text{CONT } t_1\ b\ a_1\ u\ k_1\rangle \sim \langle \text{CONT } t_2\ n\ a_2\ u\ k_2\rangle$, if

- (i) $(b,\ t_1(2)) \simeq (t_2(1) \dagger n,\ t_2(2))$, and
- (ii) $a_1 \sim a_2$ and $k_1 \sim k_2$

(7) $\langle t_1,\ b,\ v_1,\ a_1,\ u,\ k_1,\ s_1\rangle \sim \langle t_2,\ n,\ v_2,\ a_2,\ u,\ k_2,\ s_2\rangle$, if

- (i) $(b,\ t_1(2)) \simeq (t_2(1) \dagger n,\ t_2(2))$, and
- (ii) $v_1 \sim v_2$, $a_1 \sim a_2$, $k_1 \sim k_2$, and $s_1 \sim s_2$

(8) $S \sim S_F$, if S is any TBC halt state and S_F is any FBC halt state.

Table 3: Closure Conditions for $\sim$.

both *correspond* to Jumpf/False. No TBC rule corresponds to Jump. A *branch* rule means one whose name ends in /True or /False – four for TBC and two for FBC.

The definitions of $\simeq$ and $\sim$ have been constructed so that the next theorem can be proved by a straightforward, albeit lengthy argument. The reader is warned that there is a physical problem seemingly inherent in reading the body of the proof: five separate pages of this document need to be simultaneously displayed before one, namely the two tables defining $\sim$ and $\simeq$, the presentations of two rules being compared, and the argument of the proof itself. We have found no better solution than a table and an unstapler.

Theorem 10 (Preservation of State Correspondence)
Let M be a TBC *machine and M_F be a* FBC *machine with the same globals as M. Let S be the state of M and S_F the state of M_F, and assume that $S \sim S_F$. Then*

(1) if M_F proceeds by the Jump rule to state S'_F, then $S \sim S'_F$, and

(2) if M_F cannot proceed by the Jump rule, then, if either machine proceeds, then the other machine proceeds by a corresponding rule, and the resulting states correspond.

Proof. We can assume $S = \langle t\ b\ v\ a\ u\ k\ s\rangle$, and $S_F = \langle t_F\ n\ v_F\ a_F\ u\ k_F\ s_F\rangle$, where $t = \langle \texttt{template}\ b_0\ e\rangle$, and $t_F = \langle \texttt{template}\ w\ e_F\rangle$. From $S \sim S_F$ we have two facts that will be cited throughout the proof:

(i) $(b,\ e) \simeq (w \dagger n,\ e_F)$, and

(ii) $v \sim v_F$, $a \sim a_F$, $k \sim k_F$, and $s \sim s_F$.

For part (1), assume M_F can proceed by the Jump rule to state S'_F. We can thus assume that $w \dagger n = \langle \texttt{jump}\ n_1\ n_2\rangle^\frown w_1$, and we know that S'_F is the same as S_F, except that the new offset is $n + 3 + (n_1 \oplus n_2)$. Only the last clause in the definition of $\simeq$ can give (i) when $w \dagger n$ begins with jump. From the applicability of that clause it follows that

$$(b,\ e) \simeq (w_1 \dagger (n_1 \oplus n_2),\ e_F).$$

But $w_1 = (w \dagger n) \dagger 3$, so

$$w_1 \dagger (n_1 \oplus n_2) = ((w \dagger n) \dagger 3) \dagger (n_1 \oplus n_2) = w \dagger (n + 3 + (n_1 \oplus n_2)).$$

Thus we have

$$(b,\ e) \simeq (w \dagger (n + 3 + (n_1 \oplus n_2)),\ e_F).$$

Given (ii), this is all that's needed for $S \sim S'_F$. QED part (1).

For part (2), assume that M_F cannot proceed by the Jump rule. We must consider the possible applicability of each rule for each machine, but there are helpful regularities in the cases. The more complicated cases generally are those where more components of the state change, but the branch rules seem more to the point of flattening, so we start with the branch rules for M in (2A), dispense with the non-branch rules for M_F relatively easily in (2B), slog through the non-branch rules for M in (2C), and coast through the non-branch rules for M_F in (2D). Most cases are named by a specific rule of one of the machines. Such cases tacitly assume that the named rule is applicable in the appropriate machine's ingoing state.

Preservation (2A): Branch Rules for M

Case: Closed Branch/True

Assume $b = \langle\langle \texttt{unless-false}\ b_1\ b_2 \rangle\rangle$. As M_F cannot Jump, there is only one way that (i) can arise in the recursive definition of $\simeq$, namely that $w \dagger n$ is of the form $\langle \texttt{jumpf}\ n_0\ n_1 \rangle ^\frown w_1$. The definition of $\sim$ ensures that a constant on the either side is $\sim$ only to itself. We have $v \sim v_F$, so if v_F were the constant $false$ it would follow that $v = false$, which contradicts the domain conditions of the rule. Hence M_F can apply Jumpf/True.

Thus M proceeds to a state S' which is the same as S, except that its code register is b_1, and M_F proceeds to a state S'_F which is the same as S_F, except that its offset is $n + 3$. The first requirement for $S' \sim S'_F$ becomes

$$(b_1,\ e) \simeq (w \dagger (n + 3),\ e_F).$$

From (i), the form of b, and the definition of $\simeq$, we have

$$(b_1,\ e) \simeq (w_1,\ e_F).$$

But $w_1 = (w \dagger n) \dagger 3 = w \dagger (n + 3)$, and the other requirements for $S' \sim S'_F$ hold over from $S \sim S_F$, so we are done.
QED Case.

Case: Closed Branch/False

Again assume $b = \langle\langle$`unless-false` $b_1\ b_2\rangle\rangle$. As in the previous case, we can assume $w \dagger n$ is $\langle$`jumpf` $n_0\ n_1\rangle ^\frown w_1$, and can see that M_F proceeds by the corresponding rule, Jumpf/False. Only the code register of S and the offset of S_F are changed, to b_2 and $n + 3 + (n_0 \oplus n_1)$ respectively, so the resulting states correspond if

$$(b_2,\ e) \simeq (w \dagger (n+3),\ e_F).$$

From (i), the form of b, and the definition of $\simeq$, we have

$$(b_2,\ e) \simeq (w_1 \dagger (n_0 \oplus n_1), e_F).$$

But $w_1 \dagger (n_0 \oplus n_1) = w \dagger (n + 3 + (n_0 \oplus n_1))$.
QED Case.

Case: Open Branch/True

This case is similar to that for Closed Branch/True – the essential point is that the definition of $\simeq$ properly reflects the rules' difference in treatment of Closed versus Open branches.

Arguing as above, we can assume that

$$b = \langle \texttt{unless-false}\ y_1\ y_2\rangle :: b_1,$$

$$w \dagger n = \langle \texttt{jumpf}\ n_0\ n_1\rangle ^\frown w_1, \text{ and}$$

$$v \neq false \neq v_F.$$

Thus M_F proceeds by Jumpf/True. From the rules, the definition of $\sim$ and the holdover facts (ii), the only thing needed to ensure that the resulting states correspond is

$$(y_1 {}^\smile b_1,\ e) \simeq (w \dagger (n+3),\ e_F).$$

Here (i) has become

$$(\langle \texttt{unless-false}\ y_1,\ y_2\rangle :: b_1,\ e) \simeq (\langle \texttt{jumpf}\ n_0\ n_1\rangle ^\frown w_1,\ e_F).$$

Only one case of the definition of $\simeq$ can yield this, and its applicability implies that

$$(y_1 {}^\smile b_1,\ e) \simeq (w_1,\ e_F),$$

which follows, because $w_1 = (w \dagger n) \dagger 3 = w \dagger (n+3)$.
QED Case.

Case: Open Branch/False

This case is to the previous as that for Closed Branch/False is to that for Closed Branch/True. QED Case.

Preservation (2B): Branch Rules for M_F

Case: Jumpf/True

By the determinacy of pure computations and previous cases, it suffices to show is that M proceeds by one of the corresponding rules (Closed Branch/True or Open Branch/True). The two possibly applicable cases of the definition of $\simeq$ each yield the applicability of one of theses rules for M. QED Case.

Case: Inner Branch Right

Similar to the above case. QED Case.

Preservation (2C): Non-Branch Rules for M

Case: Return-Halt, Escape Halt, and Primitive CWCC-Escape-Halt

From the definition of $\sim$, if x is HALT or of the form $\langle \text{ESCAPE HALT } l \rangle$, then for all y, $x \sim y$ iff $x = y$. Then (i) and (ii) clearly imply that if any one of these rules is applicable for M then the corresponding rule is for M_F in S_F. In all cases the outgoing states are halt states of the respective machines and hence correspond. QEd Case.

Case: Return

Here we have easily that $w \dagger n$ is of the form $\langle \texttt{return} \rangle ^\frown w_1$. The second domain equation for M_F just asserts that k_F is of a certain form (actually, any non-HALT continuation). We know that k is of this form from the second domain equation for Return for M; (ii) says $k \sim k_F$, and so the definition of $\sim$ then ensures that k_F is of the required form too. So M_F can Return.

In fact, the definition of $k \sim k_F$ implies that we can write

(a) $k = \langle \text{CONT } t_1\ b_1\ a_1\ u_1\ k_1 \rangle$, and

(b) $k_F = \langle \text{CONT } t_2\ n_2\ a_2\ u_1\ k_2 \rangle$, where

(c) $(b_1,\ t_1(2)) \simeq (t_2(1) \dagger n_2,\ t_2(2))$, and

(d) $a_1 \sim a_2$ and $k_1 \sim k_2$.

The resulting states are

$$S' = \langle t_1\ b_1\ v\ a_1\ u_1\ k_1\ s\rangle \text{ and}$$

$$S'_F = \langle t_2\ n_2\ v_F\ a_2\ u_1\ k_2\ s_F\rangle.$$

The first condition on $S' \sim S'_F$ becomes just (c). The second requires that $v \sim v_F$ (which is part of (ii)), that $a_1 \sim a_2$ (part of (d)), that $k_1 \sim k_2$ (again see (d)), and that $s \sim s_F$ (part of (ii)). QED Case.

Case: Call

As in the case for Return, the first domain equation for an M_F Call follows from the first domain equation for this M Call, together with the assumption that M_F cannot Jump and the fact that $\#a = \#a_F$, which follows from $a \sim a_F$ and the definition of $\sim$.

It also follows from the applicability of Call for M that we can assume $v = \langle\text{CLOSURE}\ t_1\ u_1\ l_1\rangle$, where $t_1 = \langle\texttt{template}\ b_1\ e_1\rangle$. The definition of $v \sim v_F$ then implies that $v_F = \langle\text{CLOSURE}\ t_2\ u_1\ l_1\rangle$, where $t_1 \sim t_2$. Thus M_F can also apply Call, and the resulting states are

$$S' = \langle t_1\ b_1\ v\ a\ u_1\ k\ s\rangle \text{ and}$$

$$S'_F = \langle t_2\ 0\ v_F\ a_F\ u_1\ k_F\ s_F\rangle.$$

The first condition for their correspondence becomes

$$(b_1,\ e_1) \simeq (t_2(1) \dagger 0, t_2(2)),$$

which is equivalent to the established fact $t_1 \sim t_2$. The other conditions for $S' \sim S'_F$ amount just to (ii). QED.

Case: Escape

This case is extremely similar to the Return case, using the ESCAPE clause of the definition of $\sim$. QED Case.

Case: Make Continuation

From (i), the domain equation for Make Continuation for M, the definitions of $\simeq$ and $\sim$, and the fact that M_F can't jump, we can assume

(a) $b = \langle\langle\texttt{make-cont}\ b_0\ \#a\rangle\rangle^\frown b_1$

(b) $w \dagger n = \langle$`make-cont` $n_0\ n_1\ \#a\rangle^\frown w_1$

(c) $(b_1,\ e) \simeq (w_1,\ e_F)$, and

(d) $(b_0,\ e) \simeq (w_1 \dagger (n_0 \oplus n_1),\ e_F)$

Thus M_F can also apply Make Continuation, and we can call the resulting states S' and S'_F, where S' is the same as S except that it has code b_1, arguments $\langle\rangle$, and continuation

$$k' = \langle \text{CONT}\ t\ b_0\ a\ u\ k\rangle,$$

and S'_F is the same as S_F, except that it has offset $n+4$, arguments $\langle\rangle$, and continuation

$$k'_F = \langle \text{CONT}\ t_F\ n_2\ a_F\ u\ k_F\rangle, \text{ where}$$

$$n_2 = n + 4 + (n_0 \dagger n_1).$$

The first requirement for $S' \simeq S'_F$ becomes

$$(b_1,\ e) \simeq (w \dagger (n+4),\ e_F),$$

which follows from (c), as (b) implies that $w_1 = w \dagger (n+4)$. The second requirement becomes: $v \sim v_F$ (holds over, see (ii)); $\langle\rangle \sim \langle\rangle$ (directly from the definition of $\sim$); $k' \sim k'_F$ (see below); and $s \sim s_F$ (from (ii)). For $k' \sim k'_F$ we need $a \sim s_F$ (see (ii)), $k \sim k_F$ (see (ii)), and

$$(b_0,\ e) \simeq (w \dagger n_2, e_F),$$

which follows from (d) and $w_1 = w \dagger (n+4)$. QED Case.

Case: Other TBC Rules

All other rules follow in approximately the same way as the above ones. Several observations are in order to point out rough spots and where various wrinkles of the definitions come into play.

1. Literal, Global, and Set Global use the extra stipulation in the definition of $\simeq$ that $e(n) = e_F(n)$ (this is really just a displaced part of the code).

2. Global and Set Global use the fact that M and M_F have the same *globals*.

3. Closure needs the extra stipulations specified for its case in the definition of $\simeq$; here 2.b.iii gives exactly what is needed according to the definition of $\sim$ to conclude that the two templates of the created closures actually correspond.

4. Make Rest List needs a lemma proved by simultaneous induction, namely that if f is either of *mrl-value* or *mrl-store*, $v_1 \sim v_2$, $a_1 \sim a_2$, and $s_1 \sim s_2$, then

$$f(n,\ v_1,\ a_1,\ s_1) \sim f(n,\ v_2,\ a_2,\ s_2).$$

5. Primitive Apply-Closure and Primitive Apply-Escape both need a similar inductively established lemma about *app-stack*, namely that

$$\mathit{app\text{-}stack}(v_1,\ a_1,\ s_1) \sim \mathit{app\text{-}stack}(v_2,\ a_2,\ s_2),$$

if $v_1 \sim v_2,\ a_1 \sim a_2$, and $s_1 \sim s_2$.

QED Case.

Preservation (2D): Non-Branch Rules for M_F

By now it should be a routine matter to check that if any one of these rules is applicable, then the corresponding one is for M. Again, because of determinacy and the previous case, this is all that is needed.
QED Preservation Theorem.

8 Correctness of the Flattener

We extend the operational semantics given so far by specifying answer functions for TBC and FBC. In order to give a more general formulation of the correctness theorem, we use ternary answer functions $\mathcal{A}_{TBC}(t,\ s,\ globals)$ and $\mathcal{A}_{FBC}(t,\ s,\ globals)$, which take as arguments a template, an initial store, and a global locator function. As the last two arguments are often understood, either or both may be suppressed. We use an artificial error value so that the answer functions are total.

Given a TBC template t, store s, and global locator *globals*, let C be the unique TBC computation from the initial state

$$\langle t,\ t(1),\ \text{UNDEFINED},\ \langle\rangle,\ \text{EMPTY-ENV},\ \text{HALT},\ s\rangle.$$

If C is successful and ends with a number n in the value register, then we define $\mathcal{A}_{TBC}(t,\ s,\ globals) = n$; otherwise $\mathcal{A}_{TBC}(t,\ s,\ globals) = \bot$.

Given an FBC template t, store s, and global locator *globals*, $\mathcal{A}_{FBC}(t,\ s,\ globals)$ is defined in the same way, except starting from the initial state

$$\langle t,\ 0,\ \text{UNDEFINED},\ \langle\rangle,\ \text{EMPTY-ENV},\ \text{HALT}\ s\rangle.$$

Recall that $F(t)$ means the result of applying the flattener to a TBC template t. We need one more concept for the correctness theorem. Call a TBC store *codeless* if none of its elements begins with CLOSURE or ESCAPE. Note that a codeless TBC store is also an FBC store; for our current purposes, it is essentially flattened already, even though there is some residual tree structure (especially in the Scheme constants).

Theorem 11 Correctness of the Flattener
If t is any self-respecting TBC *template, s is any codeless store, and globals is any global locator, then*

$$\mathcal{A}_{TBC}(t,\ s,\ globals) = \mathcal{A}_{FBC}(F(t),\ s,\ globals).$$

Proof. Assuming t, s, and *globals* are as hypothesized, let H be the TBC state history generated from the initial state

$$\langle t,\ t(1),\ \text{UNDEFINED},\ \langle\rangle,\ \text{EMPTY-ENV},\ \text{HALT},\ s\rangle,$$

and H_F the FBC state history generated from the initial state

$$\langle F(t),\ 0,\ \text{UNDEFINED},\ \langle\rangle,\ \text{EMPTY-ENV},\ \text{HALT}\ s\rangle.$$

By the fact that t is self-respecting and the theorem on the establishment of $\simeq$,

$$(t(1), t(2)) \ \simeq\ (F(t(1)), F(t(2))).$$

Furthermore, $F(t) = \langle \texttt{template}\ F(t(1))\ F(t(2))\rangle$, so the first requirement for $H(0) \sim H_F(0)$ holds. The other requirements are immediate, except for $s \sim s$, which holds because s is codeless, so the initial states correspond.

Lemma 12 *For every i in the domain of H there is a j in the domain of H_F such that $i \leq j$ and $H(i) \sim H_F(j)$.*

Proof of lemma. We already have this for $i = 0$. It is enough to show that there is some $j' > j$ such that $H(i+1) \sim H_F(j')$, assuming $H(i+1)$ is defined, $i \leq j$, and $H(i) \sim H_F(j)$.

Note that an FBC computation can only apply the Jump rule finitely many times in a row, because it increases the offset register by at least three. Using this fact and the first part of the preservation theorem, we can find a j'' such that $i \leq j''$, $H(i) \sim H_F(j'')$, and the Jump rule is not applicable in state $H_F(j'')$.

Now the second part of the preservation theorem applies, and we have that $H(i+1) \sim H_F(j''+1)$ and $j''+1 > j$. QED Lemma.

If H is an infinite sequence, then H_F must be (by the $i \leq j$ part of the lemma), and both answers are $\perp$.

Assume no rule is applicable to $H(i)$, and let j be such that $H(i) \sim H_F(j)$, by the lemma. If $H(i)$ is not a halt state, then $H_F(j)$ cannot be (by the definitions of halt states and state correspondence), and both answers are $\perp$. If $H(i)$ is a halt state, then $H_F(j)$ is too, and the contents of their value registers correspond. The only thing corresponding to 0 is 0, and the only thing corresponding to 1 is 1, so the answers are equal. QED Theorem.

Note on Extensions:
We have presented TBC and FBC machines with a limited set of rules for definiteness and to save effort, but one should keep in mind that the set of rules available to the two kinds of machines can clearly be augmented without destroying the correctness theorem. It is mainly necessary to make sure that the preservation theorem is preserved, but that is easy for, say, additional primitives that might be used to manipulate strings and vectors.

References

[1] J. D. Guttman, L. G. Monk, W. M. Farmer, J. D. Ramsdell, and V. Swarup. The VLISP byte-code compiler. M 92B092, The MITRE Corporation, 1992.

[2] J. D. Guttman, L. G. Monk, J. D. Ramsdell, W. M. Farmer, and V. Swarup. A guide to VLISP, a verified programming language implementation. M 92B091, The MITRE Corporation, 1992.